Noname manuscript No.
(will be inserted by the editor)

YAMTL solution for the TTC 2021 Laboratory Workflows case

Artur Boronat

Received: June 7, 2021

Abstract In this paper, we present the YAMTL solution to the Laboratory Workflows case of TTC 2021. This solution illustrates how to specify a consistency relation between two metamodels that may map one object of the input model to several objects of the output model using a declarative style. In addition, the solution makes use of generated boilerplate code and rule inheritance for the sake of conciseness. The initial experiments show that YAMTL introduces little overhead over the reference solution, implemented in plain code using the .NET Framework, and yet it addresses its main problems: change propagation is encoded using declarative rules and traceability is handled implicitly by YAMTL.

1 Introduction

YAMTL [1] is a model transformation language for EMF models, with support for incremental execution [2], designed as an internal DSL of Xtend.

The solution to the Laboratory Workflows case illustrates novel features of YAMTL:

- Matched rules `toMany` that enable repeated rule application for the same input object subject to a valid termination condition based on the match count. The match count can be retrieved with the expression `'matchCount'.fetch()`. Whenever a rule `toMany` is involved in a rule inheritance hierarchy, all rules in that hierarchy must be `toMany` too. All variants of the operator `fetch()` have been augmented with an additional parameter, the occurrence of the transformation step from which the target object must be fetched. By default, this parameter is 0, corresponding to the first transformation step that is found for the input object or input match. Hence the `fetch` operator is equipped for working with several matches of a rule `toMany`.
- When the global correctness check, which ensures that a model transformation is a mapping, is disabled, there may be several rules transforming the same input object. This allows YAMTL to represent relational model transformations that are more expressive than mapping model transformations. The scope of the correctness check is at rule level though. A rule cannot be applied more than once to the same input object, unless the rule is declared as `toMany` and has a valid termination condition for the repetition.
- Change specifications to select the type of changes that can be propagated. Change specifications were first introduced in the Bib2Doc case of TTC 2019 where they were used to detect incorrect changes. In the solution of TTC 2021, change specifications are used to define the language of changes that need to be processed by YAMTL.
- Boilerplate code generation to reduce the amount of code needed to access matched objects and created output objects within filter expressions and output initialization actions. For example, expressions of the type `val in_sample = 'in_sample'.fetch()as Sample` can be skipped. Syntactic helpers use the names in input and output elements, so that matched objects can be accessed in filters by using the name of the corresponding input element, and both matched objects and created output objects can be accessed in output initialization actions using the name of the corresponding input/output elements. Code generation takes into account rule inheritance and appends the name of the type of the corresponding el-

A. Boronat
School of Informatics, University of Leicester, Leicester, UK
E-mail: artur.boronat@leicester.ac.uk

ement when there are ambiguities (an element is declared in different rules with different output types). Whenever, the same name has been used for an input/output element in unrelated rules and these have different types, then the name of the rule is appended to resolve ambiguities. Matched objects and built-in helpers (like `matchCount`) are also available as syntactic helpers.

2 Solution

The transformation is available at `https://github.com/arturboronat/ttc21incrementalLabWorkflows`.

In the following we briefly describe the transformation rules as they appear. Transformation rules are applied in two phases, first all labware is initialized with rules that have priority 0. Then the rest of rules are applied.

Rule `root` transforms the `JobRequest` into a `JobCollection`.

```
rule('root')
  .in('in_jobRequest', LAB.jobRequest)
  .out('out_jobCollection', JOB.jobCollection)
```

Listing 1: Rule 'root'.

Rule `reagent_trough` generates a `Trough` for each input `Reagent`.

```
rule('reagent_->_trough')
  .in('in_reagent', LAB.reagent)
  .out('out_trough', JOB.trough) [
    val in_jobRequest = in_reagent.eContainer.eContainer as
              JobRequest
    val out_jobCollection = in_jobRequest
      .fetch('out_jobCollection', 'jobRequest_->_jobCollection') as
              JobCollection
    out_trough.name = in_reagent.name
    out_jobCollection.labware.add(out_trough) ]
```

Listing 2: Rule 'reagent_trough'.

Rule `jobRequest_->_tubeRunner` generates as many `TubeRunners` as required for the input `JobRequest`, according to the expression `jobRequest.samples.size / TUBE_RUNNER_CAPACITY`. This rule has priority zero.

```
rule('jobRequest_->_tubeRunner').priority(0).toMany
  .in('jobRequest', LAB.jobRequest).filter [ matchCount <=
    max_count(jobRequest.samples.size, TUBE_RUNNER_CAPACITY)]
  .out('tubeRunner', JOB.tubeRunner)[
    val out_jobCollection = jobRequest
      .fetch('out_jobCollection', 'jobRequest_->_jobCollection') as
              JobCollection

    var tubeRunner_list = out_jobCollection.labware.filter[ it
              instanceof TubeRunner ]
    tubeRunner.name = String.format('''TubeRunner%02d''',
              tubeRunner_list.size)
    out_jobCollection.labware.add(tubeRunner) ]
```

Listing 3: Rule 'tubeRunner'.

Rule `jobRequest_->_microplate` generates as many `Microplatess` as required for the input `JobRequest`, according to the expression `jobRequest.samples.size / MICROPLATE_CAPACITY`. This rule has priority zero.

```
rule('jobRequest_->_microplate').priority(0).toMany
  .in('jobRequest', LAB.jobRequest).filter [ matchCount <=
    max_count(jobRequest.samples.size, MICROPLATE_CAPACITY) ]
  .out('microplate', JOB.microplate)[
    val out_jobCollection = jobRequest
      .fetch('out_jobCollection', 'jobRequest_->_jobCollection') as
              JobCollection
    var microplate_list = out_jobCollection.labware.filter[ it
              instanceof Microplate]
    microplate.name = String.format('''Plate%02d''',
              microplate_list.size+1)
    out_jobCollection.labware.add(microplate) ]
```

Listing 4: Rule 'microplate'.

Rule `sample_->_allocation` computes the allocation of samples to tube runner and microplate cavities. This rule is used to complete the initialization of tube runners and microplates as is therefore tagged with priority zero. The rule is transient and the `JobCollection` that is created is immaterial. Therefore, this rule is only used to perform some additional initialization in other objects. In addition, the rule remembers what sample is stored in each cavity in order to implement the application of backward changes. This rule has an undo action, which updates this backward traces when the sample is no longer allocated.

```
rule('sample_->_allocation').priority(0).transient
  .in('in_sample', LAB.sample)
    .filter [ in_sample.state == SampleState.WAITING ]
  .out('out_aux', JOB.jobCollection)[
    val in_jobRequest = in_sample.eContainer as JobRequest
    val tubeRunnerNumber = getTubeRunner_number(in_jobRequest,
              in_sample)
    val tubeRunner = in_jobRequest
      .fetch('tubeRunner', 'jobRequest_->_tubeRunner',
              tubeRunnerNumber) as TubeRunner
    tubeRunner.barcodes += in_sample.sampleID
    val microplateNumber = getMicroplate_number(in_jobRequest,
              in_sample)
    val microplateCavity = getMicroplate_cavity(in_jobRequest,
              in_sample)
    val microplate = in_jobRequest
      .fetch('microplate', 'jobRequest_->_microplate',
              microplateNumber) as Microplate
    // to facilitate backward propagation
    backward_insert(microplate.name, microplateCavity, in_sample) ]
  .undo[
    val in_jobRequest = in_sample.eContainer as JobRequest
    val microplateNumber = getMicroplate_number(in_jobRequest,
              in_sample)
    val microplateCavity = getMicroplate_cavity(in_jobRequest,
              in_sample)
    val microplate = in_jobRequest
```

```
        .fetch('microplate', 'jobRequest_->_microplate',
                 microplateNumber) as Microplate
      microplate_cavity_to_sample.get(microplate.name).remove(microplateCavity)
                 ]
```

Listing 5: Rule 'allocation'.

Rule `tip_creation` creates a `TipLiquidTransfer` for each input sample and adds itself to the corresponding `LiquidTransferJob`. When the sample has failed, the `TipLiquidTransfer` is removed in the undo action.

```
rule('tipCreation')
  .in('in_sample', LAB.sample)
    .filter[ in_sample.state != SampleState.ERROR ]
  .in('in_step', LAB.protocolStep)
    .filter[ (in_step instanceof DistributeSample ||
      in_step instanceof AddReagent)]
  .out('out_tip', JOB.tipLiquidTransfer) [
    val step = in_step
    val tip = out_tip
    val in_jobRequest = step.eContainer.eContainer as JobRequest
    val matchCount = ltjMatchCount(step, in_sample)
    val out_job = step.fetch('out_job', 'job', matchCount) as
              LiquidTransferJob
    switch(step) {
      DistributeSample: {
        tip.volume = step.volume
        out_job.source = in_jobRequest.getTubeRunner(in_sample)
        tip.sourceCavityIndex =
                    in_jobRequest.getTubeRunner_cavity(in_sample)
      }
      AddReagent: {
        tip.volume = step.volume
        tip.sourceCavityIndex = 0
        val reagent = step.reagent
        val trough = reagent.fetch() as Trough
        out_job.source = trough
      }
    }
    out_job.target = in_jobRequest.getMicroplate(in_sample)
    tip.targetCavityIndex =
              in_jobRequest.getMicroplate_cavity(in_sample)
    out_job.tips.add(tip) ]
  .undo[
    val occurrence = ltjMatchCount(in_step, in_sample)
    val out_job = in_step.fetch('out_job', 'job', occurrence) as
              LiquidTransferJob
    out_job.tips.remove(out_tip) ]
```

Listing 6: Rule 'tip_creation'.

Rule `job` is an abstract rule that declares how to create a job, adding it to the output job collection. This rule is `toMany` and all its child rules are so too.

```
rule('job').isAbstract.toMany
  .in('in_step', LAB.protocolStep)
  .out('out_job', JOB.job) [
    out_job.protocolStepName = in_step.id
    val in_jobRequest = in_step.eContainer.eContainer as JobRequest
    val out_jobCollection =
              in_jobRequest.fetch('out_jobCollection',
              'jobRequest_->_jobCollection') as JobCollection
    out_jobCollection.jobs.add(out_job)
    val prev_step = in_step.previous
    if (in_step.previous !== null) {
      val maybe_list = prev_step.fetch()
      if (maybe_list!==null) {
        if (maybe_list instanceof List) {
          out_job.previous.add(maybe_list.head)
        } else {
          out_job.previous.add(maybe_list as Job)
        }
      }
    } ]
```

Listing 7: Rule 'job'.

Rule `tipContainer` is an abstract rule that computes the number of times the rule needs to be applied for creating liquid transfer jobs. Rule `distributeSample` and `addReagent` cast down the input and output pattern elements to concrete types.

```
rule('tipContainer').isAbstract.toMany
  .inheritsFrom(#['job'])
  .in('in_step', LAB.protocolStep).filter[
    (in_step instanceof DistributeSample ||
    in_step instanceof AddReagent) &&
    matchCount <= max_count(sampleCount, MICROPLATE_CAPACITY) ]
  .out('out_job', JOB.job),

rule('distributeSample').toMany
  .inheritsFrom(#['tipContainer'])
  .in('in_step', LAB.distributeSample)
  .out('out_job', JOB.liquidTransferJob),

rule('addReagent').toMany
  .inheritsFrom(#['tipContainer'])
  .in('in_step', LAB.addReagent)
  .out('out_job', JOB.liquidTransferJob),
```

Listing 8: Rules 'tipContainer'.

Rule `plateJobs` is an abstract rule that computes the number of times the rule needs to be applied for creating jobs that work with a whole microplate (wash and incubate). Rule `wash` and `incubate` cast down the input and output pattern elements to concrete types and complete the initialization of output objects.

```
rule('plateJobs').isAbstract.toMany
  .inheritsFrom(#['job'])
  .in('in_step', LAB.protocolStep).filter[
    (in_step instanceof Wash ||
    in_step instanceof Incubate) &&
    matchCount <= max_count(sampleCount, MICROPLATE_CAPACITY) ]
  .out('out_job', JOB.job),

rule('wash').toMany
  .inheritsFrom(#['plateJobs'])
  .in('in_step', LAB.wash)
  .out('out_job', JOB.washJob) [
    val out_job = out_job_WashJob // set to vble to avoid fetching
          several times
    val microplate = getMicroplateFromMatchCount(in_step,
          matchCount)
    out_job.microplate = microplate
    val start = MICROPLATE_CAPACITY * matchCount
    val end = sampleCount - 1
```

```
      for (i: start..end)
        out_job.cavities += i % MICROPLATE_CAPACITY ],

  rule('incubate').toMany
    .inheritsFrom(#['plateJobs'])
    .in('in_step', LAB.incubate)
    .out('out_job', JOB.incubateJob) [
      val in_step = in_step_Incubate
      val out_job = out_job_IncubateJob
      out_job.temperature = in_step.temperature
      out_job.duration = in_step.duration
      val matchCount = 'matchCount'.fetch() as Integer
      val microplate = getMicroplateFromMatchCount(in_step,
            matchCount)
      out_job.microplate = microplate ]
```

Listing 9: Rules 'plateJobs'.

3 Evaluation

In the following we will consider the evaluation criteria of the article.

Understandability. The new features of YAMTL helped in specifying the transformation in a declarative way, managing traceability implicitly. Such type of specification can be challenging for model transformation languages with strong correctness criteria, where matches need to be unique: e.g. for ATL. This approach for defining the transformation should help in defining more complex behaviour in the liquid transfer robot, where additional cases are defined with additional rules. Furthermore, such additional cases can be treated using specialized rules that reuse behaviour from existing ones.

Conciseness. The transformation rules for generating jobs reuse logic, both for matching and for initializing the resulting objects, via rule inheritance. In this case, rule inheritance helped reduce the amount of code needed for each type of job. On the other hand, boilerplate code, generated by YAMTL, has helped reduce the amount of code required for fetching values from the execution environment from within filter conditions and output initialization actions.

Number of elements in the low-level model . This feature has been left for consideration at the workshop. In the given changes, all wash and incubate jobs execute without failure. It seems that this hampers the deletion of jobs, which can only occur when all of the tips of a liquid transfer job are removed. When applying changes, only changes corresponding to samples that failed (that is, whose state are error) and new samples were applied.

Execution time. For a very preliminary comparison of performance results, I implemented a correctness check algorithm[1] (following the guidelines in the case description) and a benchmark driver class. After ten runs on a MacBookPro11,5 "Core i7" 2.5 GHz with 16 GB RAM, the median run times (in milliseconds) both of the initial model transformation and of the updates are shown in Figure 1. The solutions available in the main repository were used to compare our solution.

In these results, YAMTL adds a little overhead over the reference solution and it shows a very reasonably performance for both for the initial transformation and for propagating updates forward. However, a more thorough inspection on how changes are considered for the different tools is required in order to be able to infer reliable conclusions.

References

1. Boronat, A.: Expressive and efficient model transformation with an internal dsl of xtend. In: Proceedings of the 21th ACM/IEEE International Conference on MoDELS, pp. 78–88. ACM (2018)
2. Boronat, A.: Incremental execution of rule-based model transformation. International Journal on Software Tools for Technology Transfer **1433-2787** (2020). DOI 10.1007/s10009-020-00583-y. URL `https://doi.org/10.1007/s10009-020-00583-y`

[1] The correctness check provided with the benchmark could not be executed.

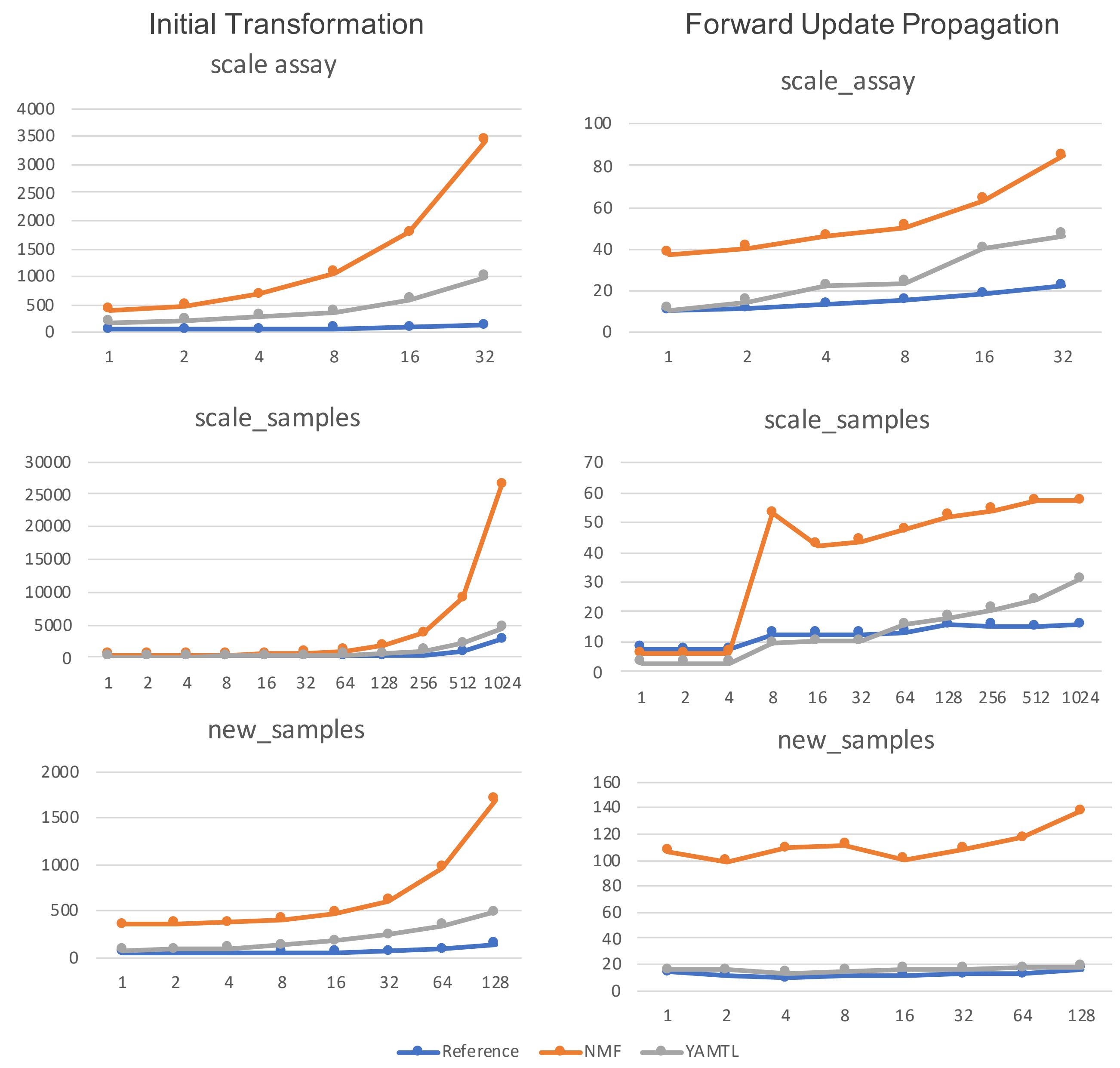

Figure 1: Preliminary results in milliseconds: initial transformation (left) and all udpates per model (right).